

## 03-可测试性：一个影响软件设计的重要因素

你好！我是郑晔。

上一讲，我们讲了软件设计的第一步：分离关注点。作为至关重要的第一步，分离关注点常常被人忽略，严重影响了设计的有效性。这一讲，我们再来看另一个经常被很多人忽视的因素：可测试性。

在讨论可测试性之前，我们不妨先来思考一个问题：你觉得软件开发中最浪费时间的环节是什么？答案肯定不是写代码，因为写代码是一个建设的过程，谈不上是在浪费时间。在我接触过的诸多项目里，集成测试可以说是一个浪费时间的大户。

那你的项目是怎么做集成测试的呢？一个常见的测试场景是这样的：你先花了一些时间打包部署一个服务端应用，然后开始测试。测着测着，你发现一个Bug，然后调查半天，最后发现是一个简单的错误。你就在心里暗恨，为啥写代码的时候没发现呢！

这还只是一个简单的场景，也有稍微复杂一点的。比如，有多个不同项目组的人一起联合测试。当你测出一个Bug，然后辛辛苦苦调查半天，发现是另外一个模块出了问题，你唯一能做的就是等着那个组的同事把Bug改好，测试才能进行下去。更可恨的是，他们查了半天，结果也是一个简单的错误。你会在心里嘀咕，为啥写代码的时候不仔细一点呢？

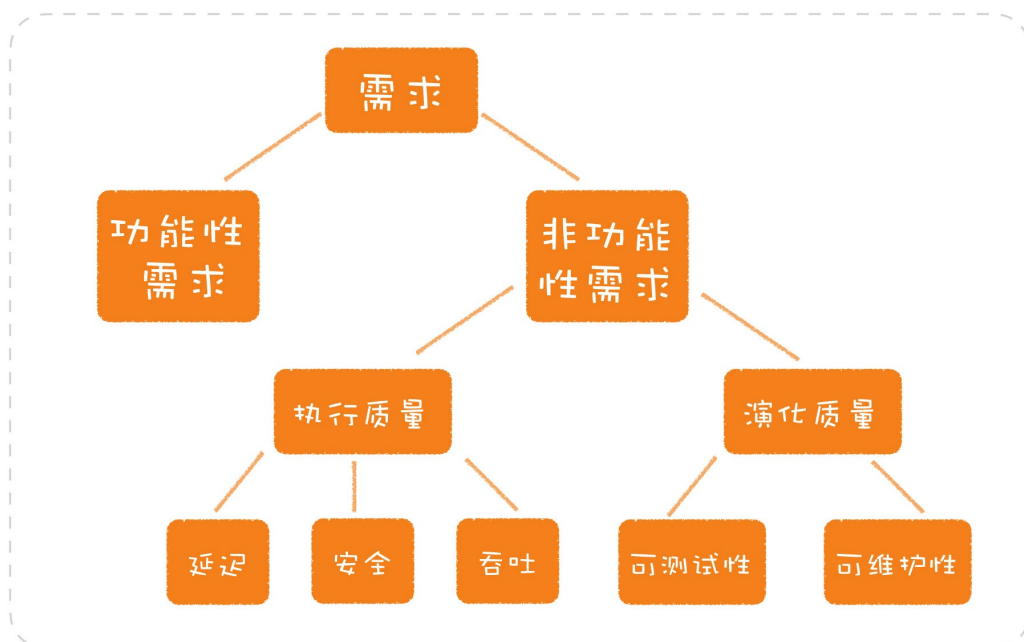
在实际工作中，我们经常遇到类似的场景。你觉得这种状态正常吗？可能很多人对此习以为常。虽然难受，却不得不忍受。

但我想说的是，这样的问题原本有机会得到优化。而出现这样的问题，主要原因就在于**前期设计时就埋下了隐患，你根本没有考虑“可测试性”**。

### 软件设计要考虑“可测试性”

我们知道，软件开发要解决的问题是从需求而来。需求包括两大类，第一类是功能性需求，也就是要完成怎样的业务功能；第二类是非功能性需求，是业务功能之外的一些需求。

非功能性需求也被分为两大类，一类称为执行质量（Execution qualities），你所熟悉的吞吐、延迟、安全就属于这一类，它们都是可以在运行时通过运维手段被观察到的；而另一类称为演化质量（Evolution qualities），它们内含于一个软件的结构之中，包括可测试性、可维护性、可扩展性等。



做设计的时候，功能性需求自不必说，你肯定会考虑到。在非功能性需求中，执行质量是很多程序员的心头爱，一般也不会被忽略。但演化质量的地位却很低，常常为人忽略，尤其是其中的“可测试性”。

我们在开发过程中欠下的很多技术债，本质上都是因为忽略了“可测试性”这个需求。

可测试性为什么如此重要？因为我们做设计，其实就是把一个软件拆分成一个一个的小模块。如果不尽可能地保证每个小模块的正确性，而只是从最外围的系统角度去验证系统的正确性，这将会是一个非常困难的过程。就和盖楼是一个道理，不保证钢筋、水泥、砖土质量合格，却想要盖出合格的大楼来，很荒唐吧！然而，很多团队的软件开发就是这么做的。

我们要保证每个小模块的正确性，就要保证每个模块在开发阶段能够测试，而想要每个模块能够测试，在设计过程中，就要保证每个模块是可以测试的，而这就是可测试性。

一旦我们在可测试性上考虑不足，就会引发一系列的后续问题。比如，复杂的系统不仅仅在测试上有难度，在集成、部署等各个环节，都有其复杂性，完成一次部署往往也需要很长时间。

这也就意味着，即便是一个简单的验证工作，部署的时间成本也非常昂贵。这还不包括在出问题时，我们在一个复杂系统中定位问题的成本。

我们只有把每个小模块尽可能做好，才能尽量降低对集成环境的依赖程度，从而节省后期的成本。这就相当于在前面多花了1块钱，却省下了后期的10块钱。

我们回过头思考一下这节课刚开始提到的那个问题，为什么我们在集成测试场景中，会浪费那么多时间呢？因为这个系统只能在集成测试环境中进行测试，所以，即使是一些非常简单的问题，也只能在这阶段暴露。这些问题原本可以在更前面的阶段解决，比如，单元测试。

可为什么这些问题会遗留到集成测试环境呢？很多程序员给你的回答都会是，不好测。而这不好测的背后，往往就是因为在设计中没有考虑“可测试性”这个因素。

那么如何在设计中考考虑可测试性呢？其实就是要在设计时想一下，**这个函数/模块/系统要怎么测。**

当你用这个标准衡量一些系统时，可能就会发现一种典型的错误，就是设计根本没有考虑过测试。这样的系统常常只有最外层的接口可以测试，也就是说，整个系统必须集成起来才能测试。前面提到的集成测试的问题犯下的就是这种错误。

在实际工作中，很多公司为了做集成测试，要把所有的子系统全部都搭建出来，也就是一套完整的环境。这种环境要占用大量的资源，一般来说，公司不会准备很多套。这样造成的结果就是各个团队对于环境的竞争，再叠加各个系统配合的问题，测试的效率还会进一步降低。

**所以，我们在设计一个函数/模块/系统时，必须将可测试性纳入考量，以便于能够完成不同层次的测试，减少对集成环境的依赖。**

那么，具体该如何做呢？一方面，尽可能地给每个模块更多的测试，使构成系统的每个模块尽可能稳定，把集成测试环境更多地留作公共的验收资源。另一方面，尽可能搭建本地的集成测试环境，周边的系统可以采用模拟服务的方案。

在软件开发过程中考虑测试，实际上是思考软件的质量问题，而把质量的思考前移到开发，甚至是设计阶段，是软件开发从传统进入到现代的重要一步。

## 当你有了可测试性的视角

现在你已经对软件设计中的可测试性有了一个初步的认识。其实，在了解可测试性之后，我们还可以把它作为一个衡量标准来考察已有的设计。

比如，有一个设计模式叫Singleton，通常的做法是把构造函数做成私有的。如果这个Singleton的类与其他组件配合，由于这个私有函数的存在，这个类无法继承，也就不能用一个子类对象去模拟它。所以，从可测试性的角度来看，Singleton就不是一个好的设计模式。

再比如，TDD（Test-Driven Development，测试驱动开发）对于很多人来说都非常困难，主要有两方面原因。一方面，这些人不习惯先写测试的工作方式，但另外一方面，也是更重要的原因，是他们不知道怎么测试。

因为很多模块的设计根本没有考虑过如何做测试，要把它们单独拿出来测试，必然会遇到很多问题。

举个例子，在通常的架构中，服务会调用数据库访问的代码。如果是不考虑测试的做法，代码可能写成这样：

```
class ProductService {
    // 访问数据库的对象
    private ProduceRepository repository = new ProductRepository();

    public Product find(final long id) {
        return this.repository.find(id);
    }
}
```

在这里，我们要直接创建数据库访问的对象，然而，要创建数据库访问对象，就要同时把数据库连接起来，

你要准备一大堆相关的东西，所以，测试的复杂度就会非常大。

可是，测试这个服务目的是，关心这个服务的逻辑是不是写正确了，这与是不是用数据库没关系啊！所以，如果我考虑了可测试性，服务的依赖就变成了一个数据访问的接口：

```
class ProductService {  
    // 访问数据库的对象  
    private ProduceRepository repository;  
  
    public ProductService(final ProduceRepository repository) {  
        this.repository = repository;  
    }  
  
    public Product find(final long id) {  
        return this.repository.find(id);  
    }  
}
```

在这种代码里，我们只需要将数据访问的接口模拟出来，而用来模拟接口的Mock框架在各种程序语言里几乎都可以找到。我们唯一要保证的，就是模拟出来的对象要与接口定义的行为保持一致，不过，这可比准备数据库，难度系数要低多了。

真正懂得了可测试性，还可以帮助我们理解软件开发的趋势。有些Java工作经验的同学可能听说过EJB（Enterprise Java Beans），它是2000年左右的开发主流。当时一个Java系统如果没用到EJB，你都不好意思和人打招呼。但是，今天你很难听说有谁还在用EJB做新系统了。

在每次测试时，EJB都需要部署到专门的应用服务器上。站在可测试性的角度看，它的测试成本就是极其高昂的，相应的开发成本也就变得很高。

当年与EJB竞争的正是当今如日中天的Spring，Spring胜出的一个重要原因就是它简化了开发。它当年的口号正是without EJB。**这是一种重要的开发趋势：轻量级开发。**而这背后，重要的思维基础，**就是可测试性**。后面在第五讲中，我们会讲到SpringDI容器的设计，你会进一步看到可测试性在其中发挥的作用。

实际上，Spring在简化开发的道路上从未停下脚步。今天的Java程序员使用Spring Boot的时候，启动它就像启动一个普通的Java应用，在IDE里做各种调试，甚至都没有注意到它启动时，下面有一个Tomcat。

要知道，当年可是要打出一个WAR包，部署到Tomcat上。所以，曾几何时，能够连接远程的Web服务器是IDE一项重要的功能，而这项功能在今天来看，已经非常鸡肋了。

## 总结时刻

今天，我们学习了一个影响软件设计的重要因素：可测试性。

在软件设计中，可测试性常常被人忽视，结果造成了很多模块的不可测，由此引发了很多技术债。所以，在设计中就要充分考虑可测试性。

在设计中考虑可测试性，就是在设计时问一下，**这个函数/模块/系统怎么测**。在软件开发中，只有把一个一

个的小模块做了足够的测试，我们才会有稳定的构造块，才可以在集成测试的时候，只关注最终的结果。

而有了可测试性的视角，我们可以把它当作一个衡量标准去看待其他的设计或实践，也可以用它帮助我们理解软件的发展趋势。

经过前几讲基础知识的铺垫，你对软件设计已经有了一个初步的了解。下一讲，我们将进入到实际的工作环节中，去了解一个软件的设计。

如果今天的内容你只能记住一件事，那请记住：**做软件设计，请考虑可测试性。**



## 思考题

最后，我想请你回想一下，如果以可测试性衡量一下你开发过的系统，它的可测试性如何？有哪些问题是由于最初没有考虑可测试性造成的呢？欢迎在留言区分享你的经历。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

## 精选留言：

- 阳仔 2020-05-29 08:27:53  
其实难度是，现在大多开发者接触的一个已有的系统，在这个系统进行维护，修改各种bug，以及添加一些新需求。那么问题来了，如何将一个已有系统改造成粒度小且可测试的程序？我觉得这个应该是大家关心的工程实践 [1赞]
- 业余爱好者 2020-05-29 01:37:48  
由于集成测试环境复杂，排错复杂，所以我们要尽量在前期，开发甚至设计阶段就要考虑测试，尽量把bug消灭在集成测试之前。每个模块保证接口功能正常，模块交互又是按照规范的，一般集成都问题不大，当然只是概率非常低。

单元测试还有一个好处，那就是自动化。写一个功能，就运行一下单元测试。单元测试是代码功能的保障。经常对曾经一些低级的bug如手误少些个！之类的错误深恶痛绝，觉得怎么能犯这么低级的错误。我错怪自己了，人就是这样一种粗心的动物，应该让程序来检查代码的正确性，程序检查的又快又不会出错，何乐不为？当然前提是得有单元测试。

Tdd是一种好的实践。在测试上要多下点功夫。  
[1赞]

- 阳仔 2020-05-29 08:22:38  
前面一节提到软件设计的首先是要分解，而分解的一个目的我觉得就是为了软件的可测试性。

当然这也是建立在分解粒度要小的基础上，

- 肥宅码农 2020-05-29 08:13:28

比如service层有个很长，包含复杂逻辑的private方法，但我想测试他。只能通过最顶层的public方法作为入口，这导致需要保证大量的前提条件的正确，我们需要mock很多外部依赖。

当private方法复杂并包含逻辑，其正确应当重构代码，而不是在测试做妥协，可以将需要测试的private方法转移到另一个对象中，成为一个public的方法。同时让我体会到测试驱动设计的含义。

- 北天魔狼 2020-05-29 06:28:47

测试前置数据，测试API，测试完清理数据。

感觉测试主要是为了防止自己或者别人改了自己的代码

- Kăfkă<sup>2020</sup> 2020-05-29 00:30:20

曾经开发过堆场应用，其中一个步骤是从远端服务器同步到本地服务器，然后再执行本地逻辑。如果每次测试本地逻辑都要从服务端拉取数据的话，就没法自动测了。当时采用的测试方法就是先抓取接口数据生成接口文件，测试就从文件中加载，再运行，最后销毁整个数据库。如果有接口相关的bug，也同样抓取数据保存，构建一个bug号命名的测试方法测试bug。

后来做过系统高可用软件，采用的方法是将代码自动部署到多个Docker里，测试代码里依据场景（为了方便，场景还用DSL写）比如杀某个Docker来测试高可用逻辑是否正常。

- Michael 2020-05-29 00:15:46

最近有同事正好在做PDF的生成，也就是把业务数据从各个别的服务拉取回来 然后清洗加工成自己想要的 数据 然后传递给模版引擎进行渲染 最终生成pdf文件上传s3 然后通过API把上传的文件地址返回给客户端 想请问老师 这部分逻辑应该怎么测试？因为同事在写完代码之后只做了简单的测试（也就是直接mock其他的service的服务然后mock数据返回 以及mock了s3上传）最后只是简单看了一下返回值是否为空就完事了。最后到环境上验证才发现好多字段的格式有问题。那像这类场景我们该怎么合理测试才能尽可能保证做到测试完整性呢？